1 Overview

The Cuyahoga eCommerce project aims to provide a flexible and extensible module for use in many different situations. Generally, we have found that very little code actually gets reused each time a store is created because requirements differ greatly each time. There is no way we could possibly cater for all eventualities, but with a few changes, possibly some custom code, and a tweak of some of the controls, we hope to cope with the majority of cases.

We aim to do that with the following strategies:

- Define simple services to hide the complexity/variation of standard use cases. In reality, most of the time we just want to browse a catalogue, add to basket, modify the basket and then pay for the item
- Exploit Castle to load custom business logic at run-time where possible
- Keep all display logic out of control code-behinds where possible. It is far easier to play with formatting in the ascx file, and does not require a recompile
- Make no assumptions about which control elements exist. If the BasketView control defines a label called `lblMessage`, we must allow users to remove that from `BasketView.ascx` without throwing an exception, within reason

1.1 Features

1.1.1 Catalogue and product repository

- Huge product repository with flexible, attribute-based product properties
- Product catalogue with navigable tree structure. Products may be attached to zero or more catalogue nodes. Multiple catalogues per store may be supported in future.
- Product cross-sell, up-sell and other product relations
- Multiple currencies supported
- Multiple languages supported (future)
- Currently supports only a single store. Further stores per Cuyahoga installation may be supported in future

1.1.2 Basket and purchasing

- Comprehensive basket system.
- Support for any tax system
- Support for any delivery system
- Basket business logic injected via a command pipeline to ease customisation
- Payment by credit card, PayPal or similar, plus trade account (future?)

2 Catalogue and product repository

2.1 Products

Products are stored within the module as base items with zero or more attributes. These attributes may be used to determine variations of the same product, such as colour, size etc, or as comparable properties between related products. All products have an item code, which may be an SKU, ISBN or other product identifier. Ideally, a textual code should distinguish between product variants; this code will vary from company to company and industry to industry. Products will have a number of associated images and thumbnails thereof. In addition, products may have related documents, technical data etc which may be accessed from product detail pages.

Cross-sell (related items such as accessories) and up-sell (more expensive/feature-rich items) will be available, allowing the vendor to increase the sale value, or number of items purchased. In addition, other related item groups may be used for other purposes.

The product detail layout will probably differ wildly from site to site so handles will be made available on the product detail page for customisers to access all of the product's properties.

2.2 Catalogue Nodes

A catalogue is a tree of nodes used to organise a set of products and allow the user to navigate and browse for items. A product may be attached to zero or more catalogue nodes, and a breadcrumb trail be made available for a user to navigate to more general items further up the tree. A catalogue node may have further information such as images and extra information. Again, there is no way to provide catalogue browsing pages that will meet all client's needs. All of the relevant catalogue properties will be available from the catalogue node pages.

2.3 ICatalogueViewService

Catalogue nodes, product details and product summaries should be accessed using an optimised service to reduce the amount of data requested by the server and increase rendering speed. Depending upon the actual performance benefit, this feature may be dropped, relying instead upon standard DAO classes as `ICatalogueViewService` is a sort of read-only DAO for catalogue nodes, products and their related entities.

3 Basket and purchasing

The basket system will perform the basic functions

- Add the current item to the basket
- View the contents of the basket
- Modify the quantities of an item
- Delete an item from the basket
- Remove all items from a basket
- Change the basket state to 'ordered'
- View the status of past orders

All basket-related functions will be accessed using `ICommerceService` to allow the simple swapping of business logic, and make testing easier. The standard implementation will use a pipeline of commands, or processors (implementing `IOrderProcessor`), to allow administrators to add business features to different stages of the basket lifetime. There will be, for example, a tax processor, a delivery charge processor, a notification processor that the customiser may choose to add to meet requirements. This will make it easy to call external systems for pricing, discount policies and the like.

The current system will only support one delivery address per order. Arguably, there may be more than one delivery address per order, or more than one order per delivery. Should there be a genuine requirement for these, it may be implemented, however, the business logic and table design could become more complex.

3.1 Tax calculation

The standard implementation of tax assigns a tax class to each product (zero-rated, tax exempt etc), and a rate for each tax region. A tax region can be assigned to a country or state. Full US taxation calculations are outside the scope of this release.

3.2 Delivery charges

Delivery charge pricing varies hugely from store to store. The standard implementation, like taxation, is assigned per zone. Future implementations may be based upon weight, size, or basket price, or a combination of all of these.

3.3 Promotions and discounts

Promotions and discounts are not part of this release, but will be implemented using the order processor classes.

3.4 Checkout stage

Since all checkout stages differ across stores, we will make a best-guess implementation using a few stages. The customer will enter their address details, fill out any further order details, then go through the payment process. A pseudo-MVC pattern may be used to navigate the user through the different stages of the order process and hopefully allow the introduction of additional stages, or removal/replacement of stages without a major rewrite of the code.

4 Administration and Configuration

The eCommerce module features a full administration console. There is a difficult balance to strike between what can/should be configured using the administration pages, and that which should be done via real coding. Since we prefer to load classes dynamically using Castle, these changes are not easily performed using normal administration.